Browne, Cameron B. (2007) Efficient Pythagorean trees: Greed is good.
Computers & Graphics 31(4):pp. 610-616.

Figures are at the end of the paper.

Efficient Pythagorean trees: greed is good

Cameron Browne

SWiSHzone.com Pty Ltd

The Basement, 33 Ewell St, Balmain, Australia, 2041

Cameron.Browne@swishzone.com

Abstract

The Pythagorean tree is a fractal figure with a branching structure based upon the Pythagorean theorem. This paper describes the use of a greedy algorithm to encourage a homogenous spread of detail for asymmetrical trees and avoid localized regions of high detail. A model of prioritized branch growth based on this idea is used to optimize tree shape, allowing more visually interesting trees to be grown using significantly fewer segments. A comparison of the fractal dimension of the resulting shapes supports the general success of this approach.

Keywords: Pythagorean tree; greedy tree; fractal; box-counting dimension.

1. Pythagorean trees

The Pythagorean tree is a plane fractal constructed with squares, such that each triplet of touching squares forms a right triangle (Figure 1). It is therefore a binary tree with the size of each parent node (*a*)

related to the sizes of the left and right child nodes (b and c) by the Pythagorean theorem $a^2 = b^2 + c^2$ [1].

Figure 2 shows the first four iterations of the Pythagorean tree for $\theta = 45^\circ$ (top row) and $\theta = 30^\circ$ (bottom row); we denote such trees by their smallest internal angle as P_{45} and P_{30}. The total number of segments s following each level of iteration i is the binomial number $s_i = 2^i - 1$ (sometimes called a Mersenne number).

If the tree is generated level by level, then the segments will be created in the breadth-first order shown in Figure 3. This order is breadth-first in the sense that from the single root node, all nodes of level 1 and grown, then from these level 1 nodes all nodes of level 2 are grown, and so on. Each segment's breadth-first order (BFO) is related to its parent's breadth-first order (BFO_p) as follows:

1) $BFO_0 = 1$ for the root node,
2) $BFO_l = 2BFO_p$ for left children, and
3) $BFO_r = 2BFO_p + 1$ for right children.

The breadth-first order indicates the distance between a given segment and the tree's root. This can be seen graphically in Figure 4, in which the segments of the first seven iterations of P_{25} are colour coded according to time of growth, so that blue represents earliest growth and red represents latest growth.

For asymmetrical trees ($\theta < 45^\circ$) the segments of latest growth vary in size across the figure, from areas of fine detail near the trunk to areas of coarse detail near the tip; this becomes increasingly pronounced

the more asymmetrical the tree. A greedy approach will now be used to achieve a more homogenous spread of detail across the figure.

2. Greedy trees

Given a set of choices in a computational problem, a *greedy algorithm* will always make the choice that appears to be the best at the time [2]. It makes a locally optimal choice in the hope that this will lead to a globally optimal solution. In the case of the Pythagoreans tree, the obvious "best choice" is to grow from the largest available segment at each step, encouraging the development of detail in more visible parts of the tree.

Figure 5 shows such a P_{25} greedy tree, which is similar to the tree shown in Figure 4 (both are grown to i=7 giving 127 segments) except that the growth order is size-based rather than breadth-first in nature. It can be seen that the greedy tree has a more homogenous spread of growth across its area at the expense of local detail, and that the points of latest growth are reasonably uniform in size. This tree is more efficient in that sense that it looks more complete for the same number of segments, and greedy growth maximizes the tree's area coverage.

Figures 6 and 7 show the standard and greedy forms of trees P_{30} and P_{15} at i=12 (8,191 segments). Note that the greedy forms are quite different in character, looking more complete and homogenous at the expense of fine detail. It can be seen that this effect is more pronounced for smaller values of θ. By contrast, the two forms of the symmetrical tree P_{45} will be identical as the left and right children of each segment will be of equal size.

3. Branch prioritization

We achieve a balance between these two extremes by prioritizing potential growth points based upon the relative importance of each. Given a linear weighting factor w that interpolates between the breadth-first (w=0) and greedy (w=1) forms, a segment's branching priority p is given by:

$$p = \frac{ws_n}{(1-w)BFO_n}$$

where s_n is the segment's size normalized to the range 0 to 1 (inclusive) based upon the size of the tree's root, and BFO_n is the segment's breadth-first order normalized to the range 0 to 1 (inclusive) based upon the total number of segments to be generated. Listing 1 shows C++ code for this weighted prioritization calculation for a given segment.

Listing 2 shows C++ code for the prioritized tree growth algorithm, given a tree's angle, greedy weighting factor w, and iteration depth i. The process starts by calculating an upper limit on the number of segments based upon i, then creating a root segment of unit size 1 and breadth-first order 1 and placing it in a priority queue (each segment is marked as a potential growth point as it is created). The highest priority segment is then removed from the front of the queue at each step, left and right children grown from it, and the children placed back in the queue according to their priority. This process is repeated until the upper limit on the number of segments is reached.

Figure 8 shows the outline of P_{30} at i=15 (65,535 segments) created a weighting factor of w=0.3 using this process. This weighting factor allows homogenous growth while maintaining a reasonable level of detail, using a relatively small number of segments.

4. Fractal dimension

Assuming that trees with a greater level of detail across more of their area will be more visually interesting, we wish to optimize the greedy weighting factor w to achieve this. There is a biological analogy here, as real trees will generally grow in such a way as to maximize the surface area of their canopy. The *fractal dimension*, a measure of how completely a fractal shape fills an area or space, provides a convenient indicator for the level of detail across a shape.

The fractal dimensions of similar tree structures have previously been calculated (for example, see [3] for detailed analyses), however these are based upon trees with regular canopies, not greedy ones. The fractal dimension D_b of the shape's outline based upon the box-counting method [1] will be used instead.

D_b is calculated by rendering the figure at a series of increasingly coarser resolutions, counting the number of pixels (boxes) that contain at least part of the figure's outline at each resolution, then finding the slope of the line-of-best-fit through a log plot of these values. This method is fast and convenient for arbitrary shapes but loses accuracy for self-intersecting figures, as areas of intersection only get counted once per iteration. For this reason, the complete outline of the figure including intersections is used rather than its silhouette (as per Figure 8), in an effort to minimize this inaccuracy.

5. Results

Table 1 shows the resulting D_b values for trees $P_{5..45}$ with various weighting factors w in the range 0 to 1 (inclusive). In each case the tree is grown to an iteration depth of i=12 for a total of 8,191 segments. This data is shown graphically in Figure 9.

As expected, D_b is generally greater for higher values of w. The D_b of P_{45} is identical for all w as the shape of P_{45} is not changed by greedy growth; this can be taken as a baseline value. The D_b of P_{40} is an anomaly as it is slightly greater for w=0 then tapers off imperceptibly as w increases, presumably due to inaccuracies in the box-counting method.

For the other trees $P_{5..35}$ it can be seen that D_b is minimal for the breadth-first form (w=0) increasing monotonically with w to be maximal for the greedy form (w=1). This effect becomes more pronounced as θ decreases, although most trees approach the baseline D_b of P_{45} as w approaches 1.

It should be pointed out that the fractal dimension of a figure actually applies to its *limiting set*, that is, its ultimate shape as its resolution approaches infinity. However, each of the measurements described above were taken at iteration depth i=12 which is essentially only a snapshot of the shape at an early stage of its potential development. The fact that higher values of w generally correspond to a greater fractal dimension indicates that for structures terminated at a finite depth (say i=12) then the greedier forms generally resemble the final form of the limiting set more closely, hence requiring fewer segments to visually approximate the figure's ultimate shape.

So what is the optimal greedy weighting w for a tree, given that lower values emphasize local detail while higher values emphasize homogeneity? Of the many trees generated, P_{30} with w=0.3 was the most

aesthetically pleasing to the author (Figure 9). It can be seen from Figure 9 that for this tree D_b reaches the baseline value and plateaus at around w=0.3, and that little advantage is gained from higher w; w=0.3 provides a good compromise between level of detail and homogeneity of detail for this tree.

Peitgen et al show a standard P_{30} tree on page 129 of their book *Chaos and Graphics* [1] grown to an iteration depth of i=26. This gives a potential total of 67,108,864 segments, although presumably most of these would have been culled as they fell below the resolution of the display device (a common technique for optimizing the display of fractals). The weighted greedy tree shown in Figure 8 compares favourably in terms of visual detail although only 65,535 segments are used, giving a potential thousand-fold saving. Weighted greedy trees allow significant savings for visually similar results, and do not require any knowledge of the display device resolution to be efficient.

6. Conclusion

A greedy approach has been used to address the lack of homogenous spread of detail across asymmetrical Pythagorean trees. The shapes of the resulting trees can be customized by prioritizing tree growth based upon a linear interpolation between the standard and greedy forms, allowing the creation of visually interesting results with significantly fewer segments. A comparison of the fractal dimension D_b of the resulting trees supports this observation in general.

Future work might include investigations into the applicability of this approach to branching structures other than Pythagorean trees, or, more generally, other than binary trees. Also, it may be warranted to

further investigate the assertion that greedy snapshots more closely resemble a figure's limiting set than non-greedy ones.

References

1. Peitgen, H., Jürgens, H. and Saupe, D. *Chaos and Fractals: New Frontiers of Science*, Springer, New York, 1992.
2. Cormen, T., Leierson, C. and Rivest, R. *Introduction to Algorithms*, MIT Press, Cambridge, 1990.
3. Mandelbrot, B. and Frame, M. The Canopy and Shortest Path in a Self-Contacting Fractal Tree, *The Mathematical Intelligencer*, 1999, 21:2, 18-27.

Code Listings

```
// Returns:
//    Branch priority in the range 0 to 1 (inclusive)
//
// Parameters:
//    w:            Greedy weighting factor 0 to 1 (inclusive) where:
//                      0 = breadth-first
//                      1 = greedy
//    upper_limit:  Maximum number of tree segments
//
double CBranch::Priority(double w, int upper_limit)
{
    return
        w * m_BaseWidth         // assume that the trunk is 1 unit wide
        +
       (1 - w) * (m_BFO / (double)upper_limit);
}
```

Listing 1. C++ code for the CBranch priority calculation.

```
// Returns:
//    Pointer to the root of the new tree
//
// Parameters:
//    a: Branch angle
//    w: Greedy weighting factor for branch prioritization
//    i: Iteration depth
//
CBranch* GrowTree(double a, double w, int i)
{
    int upper_limit = pow(2, i) - 1;     // upper limit on tree size
    CBranch* root   = new CBranch(a);    // assume width=1 and BFO=1

    CPriorityQueue queue;
    queue.Add(root);

    while (root->NumSegments() < upper_limit)
    {
        // Grow from the highest-priority branch
        CBranch* parent = queue.RemoveHead();

        CBranch* left  = parent->CreateChild(LEFT);   // sets child values
        queue.InsertInOrder( left, w, upper_limit);

        CBranch* right = parent->CreateChild(RIGHT);  // sets child values
        queue.InsertInOrder(right, w, upper_limit);
    }

    return root;
}
```

Listing 2. C++ code for the prioritized tree growth algorithm.

Tables

	***w* = 0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1**
***a* = 5**	1.062	1.205	1.245	1.271	1.282	1.296	1.304	1.312	1.319	1.335	1.349
***a* = 10**	1.135	1.247	1.265	1.277	1.288	1.304	1.320	1.341	1.368	1.399	1.428
***a* = 15**	1.213	1.302	1.323	1.342	1.357	1.369	1.381	1.394	1.410	1.427	1.447
***a* = 20**	1.316	1.379	1.396	1.409	1.417	1.426	1.432	1.439	1.449	1.461	1.472
***a* = 25**	1.369	1.438	1.450	1.458	1.463	1.468	1.472	1.476	1.481	1.487	1.493
***a* = 30**	1.422	1.468	1.475	1.480	1.483	1.485	1.487	1.491	1.494	1.498	1.501
***a* = 35**	1.460	1.480	1.483	1.485	1.486	1.487	1.488	1.489	1.490	1.491	1.493
***a* = 40**	1.503	1.496	1.497	1.497	1.497	1.497	1.500	1.501	1.501	1.501	1.502
***a* = 45**	1.483	1.483	1.483	1.483	1.483	1.483	1.483	1.483	1.483	1.483	1.483

Table 1. Box-counting fractal dimension (D_b) for Pythagorean trees of angle 5, 10, 15, ..., 45 for interpolations of w in the range 0 to 1 (inclusive).

Figure Captions

Figure 1. Construction based upon the Pythagorean theorem.

Figure 2. First four iterations of the Pythagorean trees P_{45} (top row) and P_{30} (bottom row).

Figure 3. Breadth-first growth order of the standard tree.

Figure 4. Standard tree P_{25} for i={2, 3, 4, 5, 6, 7}.

Figure 5. Greedy tree P_{25} for i={2, 3, 4, 5, 6, 7}.

Figure 6. Standard and greedy forms of P_{30} at i=12 (8,191 segments).

Figure 7. Standard and greedy forms of P_{15} at i=12 (8,191 segments).

Figure 8. Self-overlapping outline of P_{30} at i=15 (65,535 segments) for w=0.3.

Figure 9. Fractal dimension (D_b) plotted against weighting factor w for trees $P_{5, 10, 15, \ldots, 45}$.

Figures for "Efficient trees"

Cameron Browne

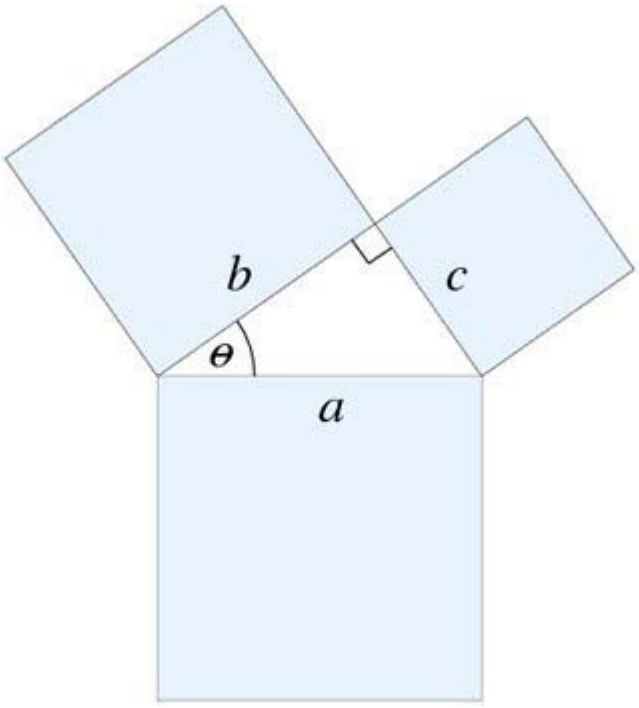

Figure 1. Construction based upon the Pythagorean theorem.

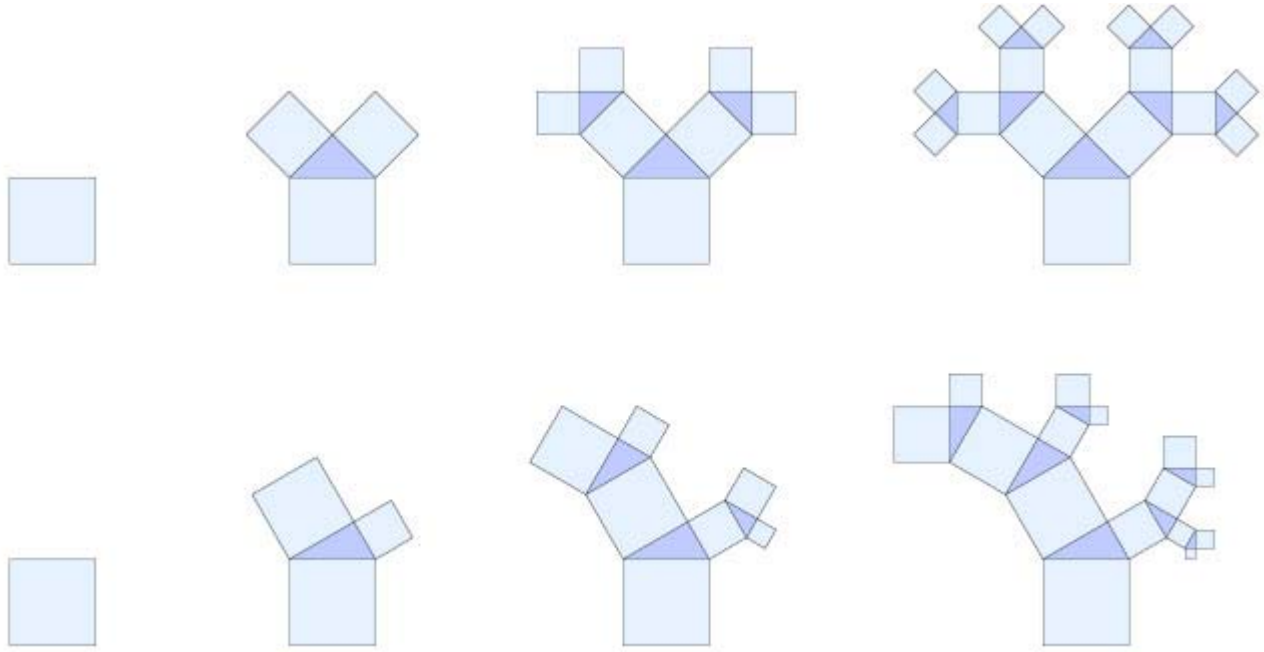

Figure 2. First four iterations of the Pythagorean trees *P*45 (top row) and *P*30 (bottom row).

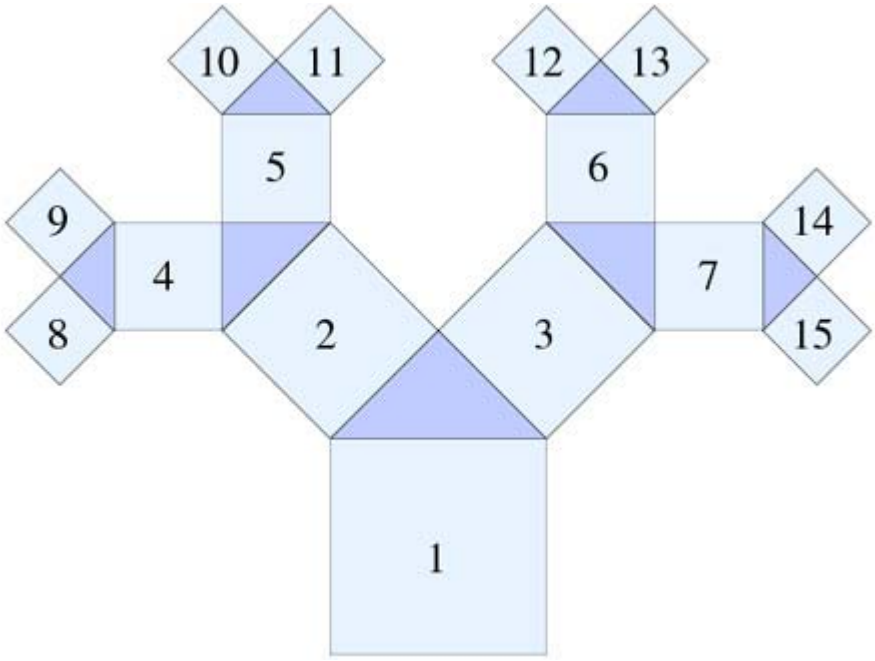

Figure 3. Breadth-first growth order of the standard tree.

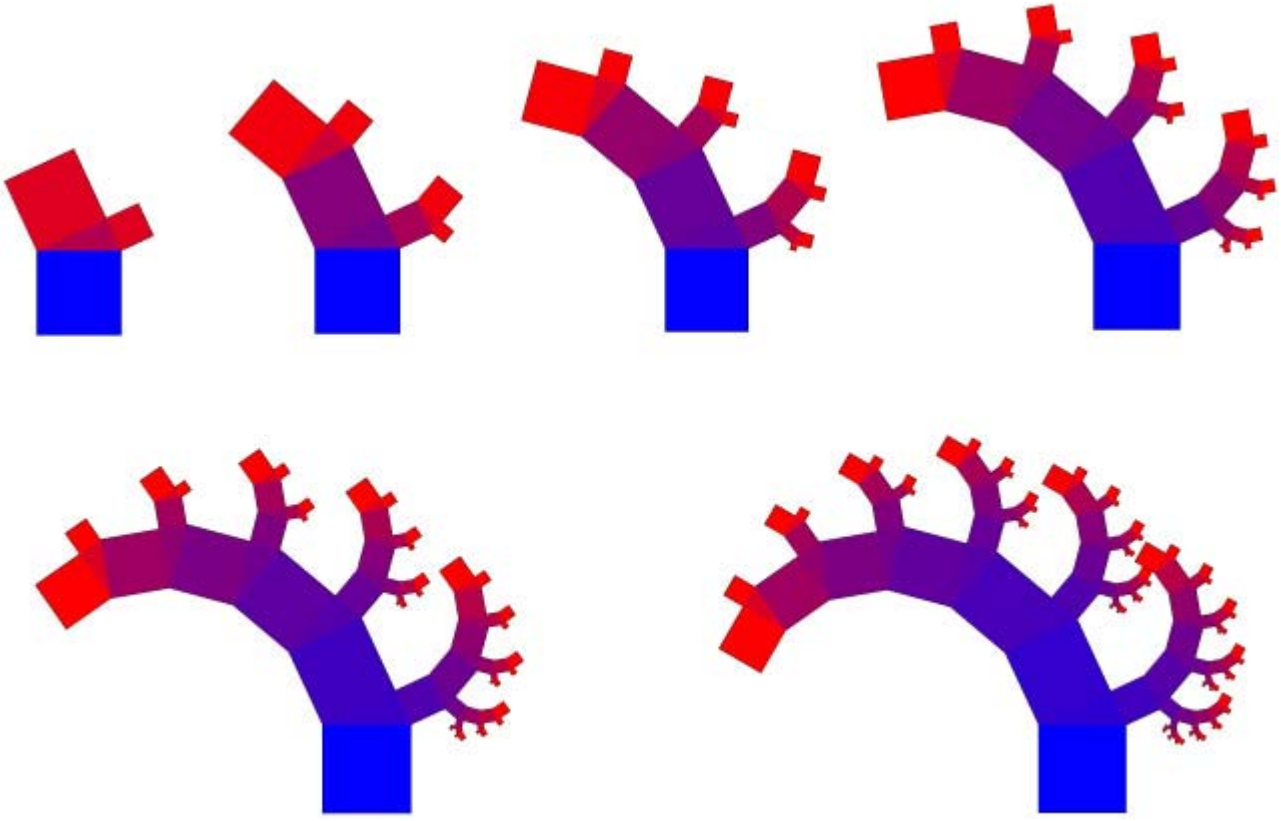

Figure 4. Standard tree *P*25 for *i*={2, 3, 4, 5, 6, 7}.

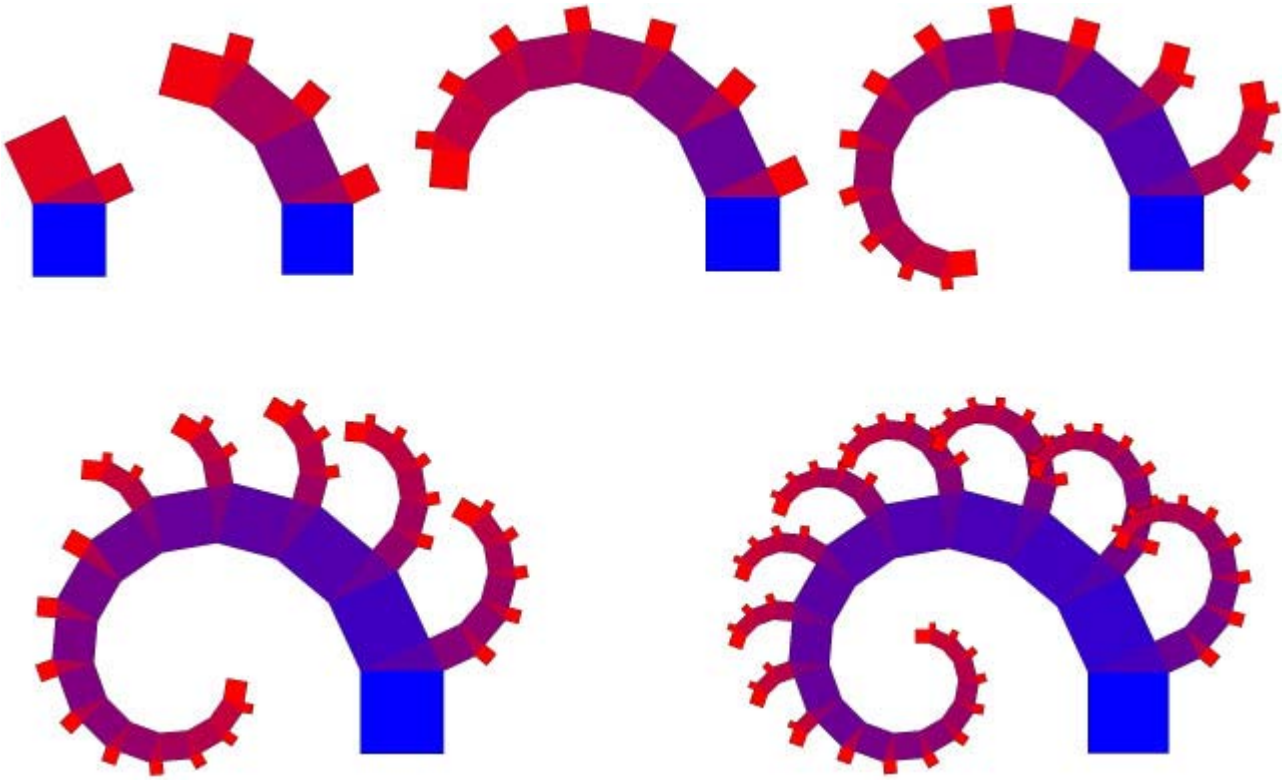

Figure 5. Greedy tree *P*25 for *i*={2, 3, 4, 5, 6,7}.

Figure 6. Standard and greedy forms of *P*30 at *i*=12 (8,191 segments).

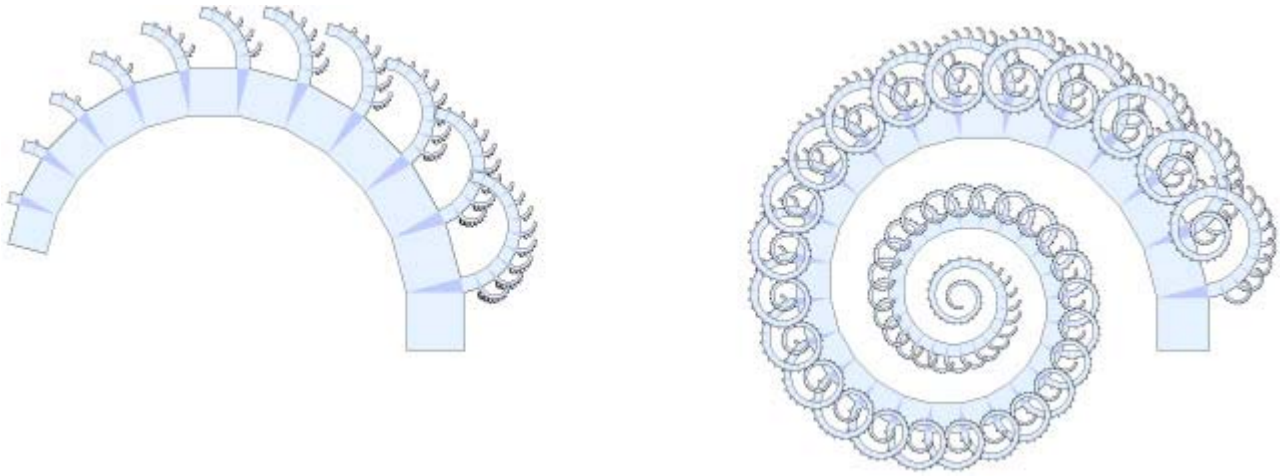

Figure 7. Standard and greedy forms of *P*15 at *i*=12 (8,191 segments).

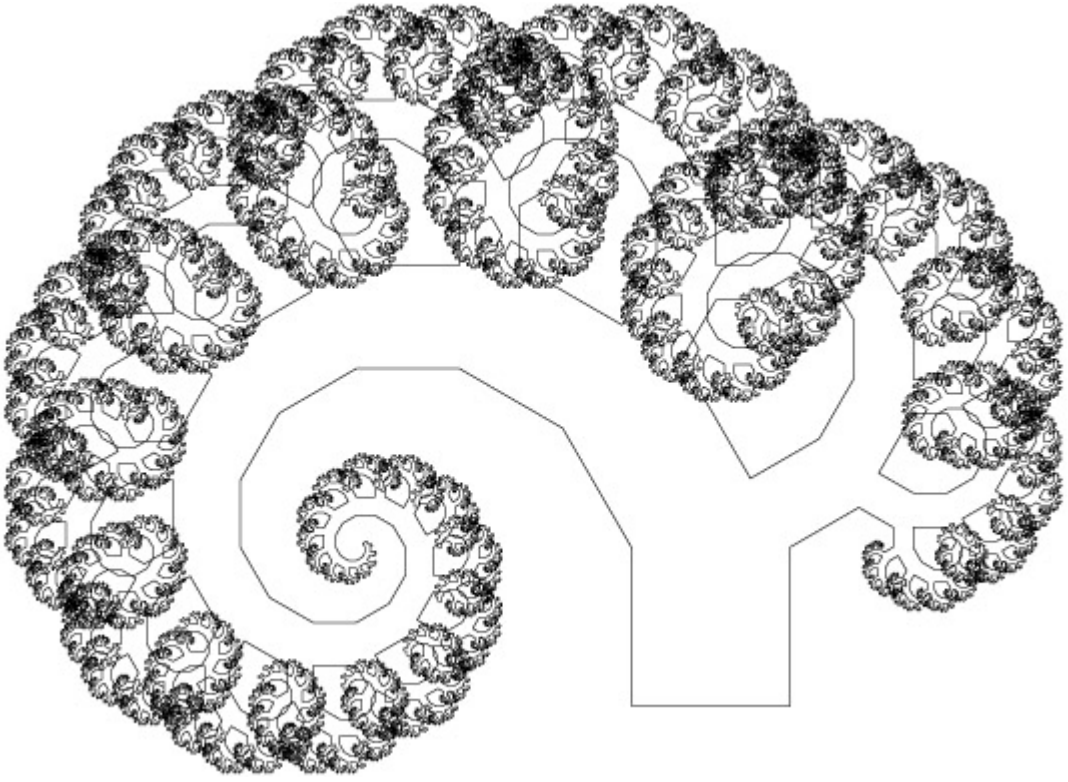

Figure 8. Self-overlapping outline of *P*30 at *i*=15 (65,535 segments) and *w*=0.3.

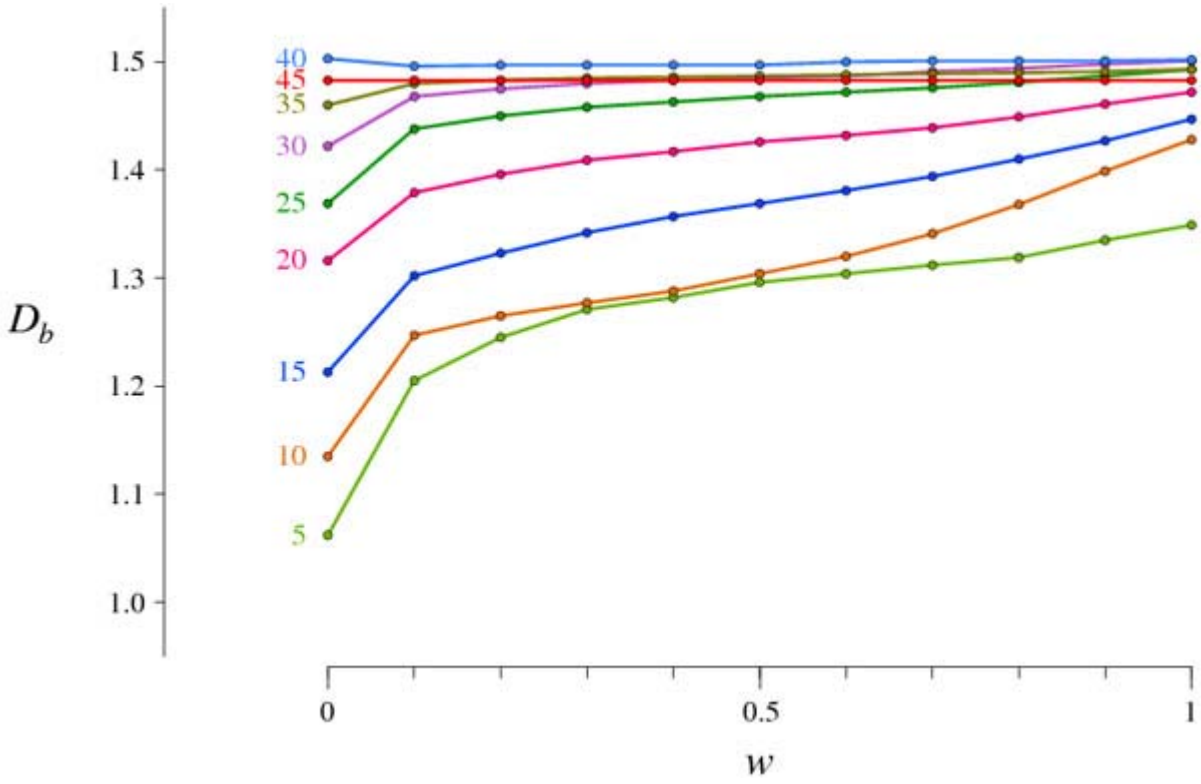

Figure 9. Fractal dimension (*Db*) plotted against weighting *w* for trees *P*5..45.